Chapter 9

The K-CHALLENGE Language

We next propose a language design scenario in which a hypothetical language designer starts with the simple imperative language in Figure 6.1 and extends it with various non-trivial language features. The purpose of this section is not to propose any novel interesting programming language (though one could write quite interesting and tricky K-CHALLENGE programs). The goal of this section is twofold:

1. To challenge the existing language definitional frameworks with a non-trivial language design task, at the same time revealing some of their inherent limitations, and
2. To show how K avoids those limitations and how it can support the proposed design scenario, requiring the designer to do minimal changes on the existing design when adding each new feature.

A major goal of an ideal language definitional framework is, of course, to support arbitrarily complex language designs with minimal burden on the user. We argue that, at least for the proposed non-trivial language design scenario, K indeed requires minimal changes on existing definitions when adding new features. We also discuss how other definitional frameworks fail to have this property. To make this language design scenario as realistic as possible, at each moment we pretend that the newly added feature is the last one to be added to the language. In other words, a feature that can be potentially added in the future cannot be used to justify a particular definitional choice at the current moment. For example, if one knew upfront that one wanted to eventually add references with explicit variable address extraction to one's language, then one may choose upfront to split the state into an environment and a store. However, we want to point out that if such a "radical" structural change requires one to revisit all or most of the existing definitions, then the underlying framework is far from ideal.

This section may indirectly also suggest that modularity of language definitions can perhaps be achieved only within a particular and well-defined universe. For example, one may devise a modular definitional methodology (e.g., a purely "syntactic" substitution-based one), where say each additional language feature is added without touching any of definitions of the previous features, in a purely functional universe known *a priori* not to allow for side effects or concurrency. If complex side effects are allowed in the universe then one may need to develop a different definitional methodology, which may also change when one adds concurrency. Moreover, the addition of new features may make previous features "obsolete" of even conflicting. For example, adding references with explicit variable address access may make the language designer prefer an assignment construct that takes a

location and a value instead of a variable and a value as before; the two cannot be both kept in the language because of conflicting semantics (if x is a variable holding a location l then one may choose $x := l'$ either to write l' in x, or to write value l' at location l). Finally, any definitional methodology developed for imperative, functional or object-oriented languages may need to be radically changed or even dropped all together when one defines a logic programming language.

Variant 1 — increment. Let us add an increment construct, $\texttt{++}\,Var$, taking a variable, incrementing its value, and then returning the new, incremented value. In K one can do it easily as follows:

$$AExp ::= \ldots \mid \texttt{++}\ Var$$
$$\langle \underline{\texttt{++}\,x}\ \cdots \rangle_k\ \langle \underline{\quad \sigma \quad} \rangle_{state} \qquad \text{where } i = \sigma[x] + 1$$
$$\quad\ i \qquad\qquad\ \sigma[i/x]$$

No change is required on the existing definition in Figure 6.1; all what needs to do is to add the syntax and the semantics of the new language construct above. Since expressions now have side effects, a big step definition of the previous variant of the language would need to be radically changed. Indeed, if expressions have no side effects, then a big-step definition can associate configurations $\langle e, \sigma \rangle$ to values v using, for example, sequents of the form $\langle e, \sigma \rangle \Downarrow v$ with the meaning "expression e evaluates in state σ to value v". Once expressions have side effects, one needs to change all the existing rules to use sequents of the form $\langle e, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$ (σ' is the state obtained after evaluating e in state σ to value v).

The other definitional styles discussed in this paper, namely (small-step) SOS, MSOS and context reduction can define increment as easily and modularly as K.

Variant 2 — merging expressions. One annoying aspect of the current language is that one can only write/read integer values in the state, meaning also that one can only assign integer type expressions to variables. We would naturally like to eliminate this limitation and add arbitrary expressions to the language, and allow them to be assigned to variables. That means, in particular that we would like to merge the different categories of expressions, currently *AExp* and *BExp*, into only one syntactic category, say $\mathcal{E}$; a type checker can be also easily defined in K to ensure that expressions are used properly, but we do not do it here. Also, we take the opportunity to add one more type of expressions, namely floats, which should, of course, enjoy the same first-class citizen rights of the other expressions. One more change to the language design is also in place here. In the original design, a program was chosen to be a statement followed by an expression. Inspired by languages such as BC, suppose that we prefer at this stage to extend the expressions with a construct "$\mathcal{E} ::= \ldots \mid Stmt; \mathcal{E}$" and to eliminate programs all together, because we can replace them with expressions. There are quite some changes above; however, they can all be done very easily in K:

- Replace *AExp*, *BExp* and *Pgm* by $\mathcal{E}$ everywhere. This can be done either by editing the existing definitions mechanically (a tedious, but automatic process), or better, by using a conventional *rename* module composition operator if the underlying implementation of K offers support for module composition. For example, with our Maude implementation of K, one can simply import the module "$\textsf{Variant1} * (\textsf{sort}\ AExp\ \textsf{to}\ \mathcal{E},\ \textsf{sort}\ BExp\ \textsf{to}\ \mathcal{E},\ \textsf{sort}\ Pgm\ \textsf{to}\ \mathcal{E})$". For example, the syntax of $\leq$ becomes "$\mathcal{E} ::= \ldots \mid \mathcal{E} \leq \mathcal{E}$" and that of $\textsf{halt}$ becomes "$Stmt ::= \ldots \mid \textsf{halt}\mathcal{E}$". All these are purely syntactic changes, with no influence on the K semantics. In fact, when using

our current implementation of K in Maude (see Section B.5), none of the existing equations or rules needs to change.

- Add boolean and float numbers to values, that is, add "$Val ::= ... \mid Bool \mid Float$", together with additional attributes (grayed below) for language constructs intended to also work with floats, namely $\_ + \_$ and $\_ \leq \_$:

$$\begin{array}{ll} \mathcal{E} + \mathcal{E} & [strict,\ extends\ \_ +_{Int}\ \_,\ extends\ \_ +_{Float}\ \_] \\ \mathcal{E} \leq \mathcal{E} & [seqstrict,\ extends\ \_ \leq_{Int}\ \_,\ extends\ \_ \leq_{Float}\ \_] \end{array}$$

The changes mentioned above are simple and necessary: the first merges the three syntactic categories into one, the second extends the strict addition to floats, and the third extends the sequentially strict less-then to floats. Any definitional style or framework must, in one way or another, do at least the above; some may need more changes than necessary. Regarding the first change, note that renaming is a well-understood module composition operator in algebraic specification supported by most algebraic specification engines, so one needs no justification for it in K. In other formalisms, including big-step/small-step SOS (modular or not) and context reduction, one may need to resort on less elegant manual (though admittedly mechanical) changes of syntactic category names, as well as of corresponding side conditions of rules. Moreover, since so far the expressions evaluated only to integer values and since less-than is *sequentially strict*, the (small-step) SOS definition would most likely contain a rule

$$\frac{\langle a, \sigma\rangle \to \langle a', \sigma'\rangle}{\langle i \leq a, \sigma\rangle \to \langle i \leq a', \sigma'\rangle}, \quad \text{where } i \in Int,\ a, a' \in AExp,\ \sigma, \sigma' \in State.$$

Unless one envisioned such possible language extensions upfront and defined the rule above for any values $v \in Val$ instead of for any integer $i \in Int$, similar rules need to be added for each extension, in particular

$$\frac{\langle a, \sigma\rangle \to \langle a', \sigma'\rangle}{\langle f \leq a, \sigma\rangle \to \langle f \leq a', \sigma'\rangle}, \quad \text{where } f \in Float,\ a, a' \in AExp,\ \sigma, \sigma' \in State.$$

Similarly, if in a context reduction definition of this language the sequential strictness of "$\leq$" was defined using a production "$Cxt ::= ... \mid Int \leq Cxt$" like in Figure 1, then one needs to add one more production to evaluation contexts, namely "$Cxt ::= ... \mid Float \leq Cxt$".

Note that adding such artifact rules or declarations is not necessary in K, because the *seqstrict* attribute is defined in terms of computations (K) and result computations (*KResult*), and that values were already defined to be computation results (a necessary step when using the *seqstrict* attribute). In fact, our major reason for introducing the strictness attributes was precisely to eliminate the need to introduce or revisit such low-level and uninteresting rules. The user of K needs to think and define the intended evaluation strategy of each language construct once and for all (right after defining its syntax). Thinking and designing a language in terms of computations and computation transformations, as opposed to particular syntactic categories and syntactic transformations, brings in our view a level of abstraction that enhances the modularity of the definitional framework.

---

start my comment

A Simple Type Checker

$\mathit{TypeExp} ::= \mathit{int} \mid \mathit{float} \mid \mathit{bool}$, $\mathcal{T} ::= \mathit{TypeExp} \mid \mathit{stmt}$	$x.\mathsf{int} \to \mathit{int}$, $x.\mathsf{float} \to \mathit{float}$, $x.\mathsf{bool} \to \mathit{bool}$,
$\mathcal{E}, \mathit{Stmt}, \mathcal{T} \leq K$	$\mathsf{++}\, x.\mathsf{int} \to \mathit{int}$, $\mathit{int} + \mathit{int} \to \mathit{int}$, $\mathit{float} + \mathit{float} \to \mathit{float}$,
$\mathit{Config} ::= \mathit{TypeExp} \mid [\![K]\!]_\tau$, $[\![\mathit{texp}]\!]_\tau = \mathit{texp}$	$\mathit{int} \leq \mathit{int} \to \mathit{bool}$, $\mathit{float} \leq \mathit{float} \to \mathit{bool}$,
$\_+\_, \_\leq\_, \_\mathsf{and}\_, \mathsf{not}\_$ $[\mathit{strict}]$	$\mathsf{not}\, \mathit{bool} \to \mathit{bool}$, $\mathit{bool}\, \mathsf{and}\, \mathit{bool} \to \mathit{bool}$,
$\_:=\_, \mathsf{if}\_\_\_, \mathsf{while}\_\_, \mathsf{halt}\_$ $[\mathit{strict}]$	$\tau := \tau \to \mathit{stmt}$, $\mathsf{if}\, \mathit{bool}\, \mathit{stmt}\, \mathit{stmt} \to \mathit{stmt}$,
$\_;\_$ $[\mathit{strict}(1)\ \_]$	$\mathsf{while}\, \mathit{bool}\, \mathit{stmt} \to \mathit{stmt}$, $\mathsf{halt}\, \mathit{texp} \to \mathit{stmt}$

---

end my comment

Variation 3 – output. Let us now add output to our language, that is, a statement "$\mathsf{output}\ \mathcal{E}$" taking an expression, evaluating it, and then outputting its value into a buffer (i.e., list) that collects all the output. In addition to the strict language construct "$\mathsf{output}\ \mathcal{E}$", the configuration also needs to incorporate an output buffer, say wrapped by configuration item label *output*. Since values can be now collected in the output buffer, there is no need for a program to evaluate or to halt to a value; in other words, one can pass to the "$[\![\_]\!]$" operation a statement computation and modify the configuration initialization and termination, as well as the syntax and semantics of halt, accordingly (halt takes no arguments now, so we remove its previous syntax/semantics and add the following instead):

$$\mathit{Stmt} ::= \ldots \mid \mathsf{output}\_ \quad [\mathit{strict}] \mid \mathsf{halt}$$
$$\mathit{Config} ::= \ldots \mid \mathsf{List}[\mathit{Val}] \mid \langle \mathsf{List}[\mathit{Val}] \rangle_{\mathit{output}}$$

$$[\![s]\!] = \langle \langle s \rangle_k\ \langle \cdot \rangle_{\mathit{state}}\ \langle \cdot \rangle_{\mathit{output}} \rangle_\top$$
$$\langle \cdots\ \langle \cdot \rangle_k\ \langle \mathit{vl} \rangle_{\mathit{output}}\ \cdots \rangle_\top = \mathit{vl}$$
$$\langle \mathsf{halt}\ \cdots \rangle_k \to \langle \cdot \rangle_k$$

$$\langle \underset{\cdot}{\underline{\mathsf{output}\ v}}\ \cdots \rangle_k\ \langle \cdots\ \underset{v}{\underline{\cdot}} \rangle_{\mathit{output}}$$

We claim that the K definition above is minimal. Indeed, the two declarations for halt (new syntax and rule) were necessary because we decided for a completely different halt statement. Also, each of the remaining six declarations above states a different and necessary part of the semantics of output: the first declares its syntax and evaluation strategy, the second declares the buffer in which the output values are collected, the third defines the new structure of the configuration that accommodates evaluations of statements and lists of values as results, the fourth and the fifth define the initialization and the termination of the computation using the new configuration structure, while the sixth gives the actual semantics of output.

Adding output to a language defined using conventional big-step or small-step SOS is devastating: one needs to change every single rule to accommodate the new configuration containing the output buffer in addition to the syntax and the state. Like K, MSOS elegantly avoids doing that; all what needs to do in MSOS is to add an output label on transitions that is only used in the semantics of output. Context reduction is more modular than SOS, but one's skill plays a more important role in achieving overall modularity. For example, the immediate way to add output "modularly"

in a context reduction definition is to change the context production "$Cxt ::= ... \mid [\![Cxt, State]\!]$" into "$Cxt ::= ... \mid [\![Cxt, State, Output]\!]$". None of the other productions need to change and so do all the existing reduction rules that do not mention the configuration construct, e.g., "$\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2$", etc. Unfortunately, as an artifact of how matching works in context reduction, rules that use the configuration need to change to accommodate the new configuration. For example, the rule "$[\![c, \sigma]\!][x] \to [\![c, \sigma]\!][\sigma[x]]$" needs to change to "$[\![c, \sigma, o]\!][x] \to [\![c, \sigma, o]\!][\sigma[x]]$", even though the new addition to the configuration, the output, plays no role in the semantics of variable lookup. The "user's skill" we mentioned above to make context reduction also modular in this case, is to envision possible configuration changes and thus define a configuration as a list structure containing various configuration items, for example:

$$\begin{array}{rcl} \mathit{ConfigItem} & ::= & \mathit{State} \mid \mathit{Output} \mid ... \\ \mathit{Cxt} & ::= & [\![\mathit{Cxt}, \mathit{ConfigItem}^*]\!] \end{array}$$

Then one can write the reduction rules to match in the list of configuration items only those items of interest, for example ($\gamma, \gamma' \in \mathit{ConfigItem}^*$, $c \in \mathit{Cxt}$, $\sigma \in \mathit{State}$, $x \in \mathit{Var}$, $o \in \mathit{Output}$, $v \in \mathit{Val}$):

$$[\![c, (\gamma, \sigma, \gamma')]\!][x] \to [\![c, (\gamma, \sigma, \gamma')]\!][\sigma[x]]$$
$$[\![c, (\gamma, o, \gamma')]\!][\mathsf{output}\ v] \to [\![c, (\gamma, (o, v), \gamma')]\!][\mathsf{skip}]$$

Even though one still needs to mention irrelevant variables just for matching reasons, such as the lists of configuration items γ and γ', the context reduction definition becomes with this change much more modular than before. From a K perspective, a slight inconvenience in both MSOS and context reduction is that one needs to introduce (if not already in the language) the "value" statement skip and then, using an additional reduction step, discard it; in K one just dissolves a statement once finished, thus capturing the intended computation granularity of the statement construct. Introducing skip and changing the computation granularity of statements is, however, generally accepted by purely syntactic definitional approaches.

Variation 4(a) – λ with substitution. The language defined so far has no capabilities to group code in functions or procedures. We next enrich our language with λ-expressions; more precisely, we add λ-abstraction and λ-application as new expression constructs and then give them a substitution-based semantics. Any of the variants of λ-calculus discussed in Section 6.0.11 can be considered. For the sake of concreteness we, however, choose the call-by-value parameter passing style here:

$$\begin{array}{l} \mathit{Val} ::= \ldots \mid \lambda\, \mathit{Var}.\mathcal{E} \\ \mathcal{E} ::= \ldots \mid \mathcal{E}\ \mathcal{E} \quad [\mathit{strict}] \end{array}$$

$$\frac{\langle (\lambda x.e)\ v\ \cdots \rangle_k}{e[v/x]} \quad \text{where } x \in \mathit{Var},\ e \in \mathcal{E},\ v \in \mathit{Val}.$$

One (rather standard) problem with definitions like the above in the context of λ-calculus with imperative features, is that one cannot assign the bound variable (x) in the body of the λ-abstraction (e), so one may need a static analysis to discard programs that do it. A (rather standard) alternative to this is to instead introduce a fresh variable, say y, bind it to the argument value (v) in the state, and then replace the bound variable x by y. We can do all these in one step in K, replacing the rule above with the following one:

$\langle \underline{(\lambda x.e)\, v} \cdots \rangle_k \; \langle \underline{\quad \sigma \quad} \rangle_{state}$ where $y \in Var$ is a fresh name.
$e[y/x]$ $\sigma[v/y]$

The reason we need a fresh name (y) and an α-conversion to replace the bound name (x) in the rule above instead of just binding x to v in the state, is because x may already be used outside the scope of the λ-abstraction and we obviously do not want to affect its value there.

Context reduction definitions take a similar approach to add the features above. When adopting a substitution-based style as above (which may not necessarily be always possible or the best, e.g., when adding references and concurrency to the language), a slight advantage of SOS approaches is that they do not need a fresh name: instead, they just initiate in the condition of the rule a reduction of e in the *modified* state $\sigma[v/x]$, and then continue the reduction in the conclusion of the rule using the original state, σ.

Now that binding is available, the language designer may want to disallow reads and writes of variables that were not explicitly bound. From here on, we therefore assume that the original expressions/program to evaluate is closed (this can be easily achieved automatically by wrapping the entire program in a series of λ abstractions binding all the originally free variables and then calling it on initial values for those variables).

---

add call by need?

---

prove correctness of call by need

Variation 4(b) – λ with closures

Suppose that, for several reasons, our language designer decides to switch to an environment-based definition of her language. As usual, we assume an infinite number of possible locations and that we can get fresh locations whenever needed. A major structural change in the configuration is required, namely the state is split into an environment and a store. This structural change inevitably affects the existing rules that relied on the state (i.e., explicitly matched it), but, fortunately, the other rules need not be touched. Below we only list those rules that changed, mentioning that we take a liberty here to use the *restore* computation item also used in Section 6.0.20 which is builtin in our implementation of K (see Appendix B.6):

$$Config ::= \langle K \rangle_k \mid \langle Env \rangle_{env} \mid \langle Store \rangle_{store} \mid \langle \mathsf{List}[Val] \rangle_{output} \mid \langle \mathsf{Set}[Config] \rangle_{\top}$$
$$[\![e]\!] = \langle \langle e \rangle_k \ \langle \cdot \rangle_{env} \ \langle \cdot \rangle_{store} \ \langle \cdot \rangle_{output} \rangle_{\top}$$

$$\langle \frac{x}{\sigma[\rho[x]]} \cdots \rangle_k \ \langle \rho \rangle_{env} \ \langle \sigma \rangle_{store}$$

$$\langle \frac{x := v}{\cdot} \cdots \rangle_k \ \langle \rho \rangle_{env} \ \langle \frac{\sigma}{\sigma[v/\rho[x]]} \rangle_{store}$$

$$\langle \frac{\mathsf{++}\, x}{i} \cdots \rangle_k \ \langle \rho \rangle_{env} \ \langle \frac{\sigma}{\sigma[i/\rho[x]]} \rangle_{store} \quad \text{where } i \text{ is } \sigma[\rho[x]] + 1$$

$$Val ::= \ldots \mid closure(Var, \mathcal{E}, Env)$$
$$\mathcal{E} ::= \ldots \mid \lambda\, Var.\mathcal{E} \mid \mathcal{E}\ \mathcal{E} \ [strict]$$

$$\langle \frac{\lambda x.e}{closure(x, e, \rho)} \cdots \rangle_k \ \langle \rho \rangle_{env}$$

$$\langle \frac{closure(x, e, \rho)\ v}{e \curvearrowright restore(\rho')} \cdots \rangle_k \ \langle \frac{\rho'}{\rho[l/x]} \rangle_{env} \ \langle \frac{\sigma}{\sigma[v/l]} \rangle_{store} \quad \text{where } l \text{ is a fresh location}$$

Variant 5 – recursion. Let us next add an explicit construct for recursion, namely μ:

$$\mathcal{E} ::= \ldots \mid \mu\, Var.\mathcal{E}$$
$$\langle \mu x.e \cdots \rangle_k = \langle (\lambda x.e)\, (\mu x.e) \cdots \rangle_k$$

One could also give μ a direct semantics, not relying on λ, but one would have to add a closure value like we did for the semantics of λ. Note that we need not worry about affecting the intended computation granularity of μ with our definition above, because we used an equation, not a rule. Other formalisms may be forced to give μ a direct semantics if unaffected computation granularity is important to the designer.

---

prove formally that the two definition of μ are indeed equivalent

---

Variant 6 – references. Let us next add references to the language, together with dereferencing and explicit address extraction for names. These suggest one major change in the design of the language: the assignment statement can be changed to assign values to locations, instead of to variables. To do so, one also needs to extend the values with locations and re-refine the assignment to be strict in both its arguments. Here are all the changes needed to the existing language in order to incorporate the above (one also needs to remove the annotated syntax for assignment and its rule from the previous definition):

$Val ::= \ldots \mid Loc$
$\mathcal{E} ::= \ldots \mid \mathsf{ref}\,\mathcal{E}\ [strict] \mid *\,\mathcal{E}\ [strict] \mid \&\ Var$
$Stmt ::= \ldots \mid \mathcal{E} := \mathcal{E}\ [strict]$

$\langle \frac{\mathsf{ref}\, v \ \cdots}{l} \rangle_k\ \langle \frac{\sigma}{\sigma[v/l]} \rangle_{store}$ where l is a fresh location

$\langle \frac{*\, l \ \cdots}{\sigma[l]} \rangle_k\ \langle \sigma \rangle_{store}$

$\langle \frac{\&\, x \ \cdots}{\rho[x]} \rangle_k\ \langle \rho \rangle_{env}$

$\langle \frac{l := v \ \cdots}{\cdot} \rangle_k\ \langle \frac{\sigma}{\sigma[v/l]} \rangle_{store}$

Peter: should := be seqstrict

Variant 7 – call with current continuation. To add call/cc to the language, all one needs to do is to add the following without changing anything to the existing definitions:

$\mathcal{E} ::= \ldots \mid \mathsf{callcc}\,\mathcal{E}\ [strict]$
$Val ::= \ldots \mid cc(K, Env)$

$\langle \frac{\mathsf{callcc}\, v \ \curvearrowright k}{v\ cc(k,\rho)} \rangle_k\ \langle \rho \rangle_{env}$

$\langle \frac{cc(k,\rho)\, v \curvearrowright \_}{v \curvearrowright k} \rangle_k\ \langle \frac{\_}{\rho} \rangle_{env}$

Variant 8 – nondeterminism. Recall that one of our major goals was to design a language definitional framework that is executable. Nondeterminism is trivial to add to all non-functional/non-denotational definitional approaches. Here is our definition of a trivial non-deterministic random boolean choice:

$\mathcal{E} ::= \ldots \mid \mathsf{randomBool}$
$\mathsf{randomBool} \to \mathsf{true}$
$\mathsf{randomBool} \to \mathsf{false}$

While functional or denotational approaches also give support for nondeterminism, these approaches have a different goal than ours: to capture *all* possible behaviors of a nondeterministic program as a value. Needless to say that that operation is very expensive and, sometimes impossible: for example when a program has an infinite number of behaviors. Our goal is to get executable semantics, so that programs with infinite behaviors can still be executed, with the possibility to *also* obtain all possible

behaviors if one is interested in that (using, e.g., a search command in our Maude implementation of K — Appendix B.5).

Variant 9 – aspects. There are many approaches to aspects and aspect-oriented programming these days. We, of course, do not intend to cover all those here. We only want to show how easily and modularly one can add aspects to a language formally defined in K. To illustrate this point, we consider only one language construct, "$\mathsf{aspect}\ s$", taking a statement and executing it whenever a function is being called from from there on. The aspect statement is executed "as is", in the sense that aspects come with no static scoping for the variables that they may access or change. It is easy to modify our definition to assume statically scoped aspects, but that is not our purpose here. We allow the aspect to be dynamically changed during the execution of the program. To achieve that, we define one more configuration cell that keeps the aspect:

$$\begin{array}{l} \mathit{Stmt} ::= \ldots \mid \mathsf{aspect}\ \mathit{Stmt} \\ \mathit{Config} ::= \ldots \mid \langle K \rangle_{\mathit{aspect}} \end{array}$$

$$[\![s]\!] = \langle \langle s \rangle_k\ \langle \cdot \rangle_{\mathit{env}}\ \langle \cdot \rangle_{\mathit{store}}\ \langle \cdot \rangle_{\mathit{output}}\ \langle \cdot \rangle_{\mathit{aspect}} \rangle_\top$$

$$\langle \underset{\cdot}{\underline{\mathsf{aspect}\ s}} \cdots \rangle_k\ \langle \underset{s}{\underline{\_}} \rangle_{\mathit{aspect}}$$

There are several semantic choices possible, aspect-oriented frameworks opting for one or more of them. For example, one possibility is to grab the aspect available at function declaration time, another is grab it at function application time. Also, for any of these two possibilities, the aspect can be executed in caller's context or in callee's context. Any of these can be easily defined in K. For example, here is a definition in which the aspect is grabbed at function declaration time and later executed in callee's context:

$$\langle \underset{\mathit{closure}(x, (s \curvearrowright e), \rho)}{\underline{\lambda x.e}} \cdots \rangle_k\ \langle \rho \rangle_{\mathit{env}}\ \langle s \rangle_{\mathit{aspect}}$$

and here is a definition in which the aspect is executed in callee's context at function invocation time:

$$\langle \underset{s \curvearrowright e \curvearrowright \mathit{restore}(\rho')}{\underline{\mathit{closure}(x, e, \rho)\ v}} \cdots \rangle_k\ \langle \underset{\rho[l/x]}{\underline{\rho'}} \rangle_{\mathit{env}}\ \langle \underset{\sigma[v/l]}{\underline{\sigma}} \rangle_{\mathit{store}}\ \langle s \rangle_{\mathit{aspect}} \quad \text{(where } l \text{ is a fresh location)}$$

Variant 10 – concurrency with lock synchronization. Let us next extend this language with dynamic threads that can run concurrently. Whether multi-threading with shared memory is how concurrency should be supported in languages is an interesting subject open to debate (where we have our personal and therefore subjective opinions, but here we refrain from commenting on this subject). Wearing the hat of the designer of a language design framework (and not that of a programming language designer), our position here is that a powerful language design framework should support all existing approaches to concurrency; at the time this paper was written (end of 2007) multi-threading with shared memory was still the most practical approach to concurrency.

We consider the following additional syntax:

$$\mathit{Stmt} ::= \ldots \mid \mathsf{spawn}\ \mathit{Stmt} \mid \mathsf{acquire}\ \mathcal{E}\ [\mathit{strict}] \mid \mathsf{release}\ \mathcal{E}\ [\mathit{strict}]$$

spawn s spawns a new thread executing statement s concurrently with the rest of the threads. The newly created thread inherits the environment of its parent at creation time, which is the means by which memory is shared, and dissolves itself when the statement s is completely processed. Threads synchronize through acquiring and releasing locks. Any values can be used as locks, including functions, and two locks are considered equal if and only if they are *identical* as value terms (not even α-equivalent). Locks can be acquired and released multiple times by the same thread and only one thread can hold a lock at any given time. A lock is effectively released by a thread only when the number of releases matches the number of acquires. If a lock is not available (because it is taken by another thread), then the thread attempting to acquire it waits until the lock is released.

We modify the configuration so that the information pertaining to each thread (i.e., computation, environment and lock/counter pairs) is held into one cell of type *thread*. An additional top-level cell is needed to hold all the busy locks. The semantics of thread termination is to dissolve its corresponding thread cell, releasing all its resources. Therefore, the execution of a program is terminated when there is no thread cell left. Here is the new configuration structure, its initialization and its termination:

$$\begin{array}{l} \mathit{Config} ::= \ldots \mid \langle \mathsf{Set}[\mathit{Val} \times \mathit{Nat}] \rangle_{\mathit{holds}} \mid \langle \mathsf{Set}[\mathit{Config}] \rangle_{\mathit{thread}} \mid \langle \mathsf{Set}[\mathit{Val}] \rangle_{\mathit{busy}} \\ [\![s]\!] = \langle \langle \langle s \rangle_k \ \langle \cdot \rangle_{\mathit{env}} \ \langle \cdot \rangle_{\mathit{holds}} \rangle_{\mathit{thread}} \ \langle \cdot \rangle_{\mathit{store}} \ \langle \cdot \rangle_{\mathit{output}} \ \langle \cdot \rangle_{\mathit{aspect}} \ \langle \cdot \rangle_{\mathit{busy}} \rangle_{\top} \\ \langle \langle \_ \rangle_{\mathit{store}} \ \langle \mathit{vl} \rangle_{\mathit{output}} \ \langle \_ \rangle_{\mathit{aspect}} \ \langle \_ \rangle_{\mathit{busy}} \rangle_{\top} = \mathit{vl} \end{array}$$

Before we attempt to give the semantics of the concurrency-related language constructs, some explanations are needed with regards to the change of the configuration structure. A major problem when changing the structure of the configuration is that some of the previously defined rules and equations may not match anymore against the new configuration. Indeed, consider for example the rule for variable lookup:

$$\langle \frac{x}{\sigma[\rho[x]]} \cdots \rangle_k \ \langle \rho \rangle_{\mathit{env}} \ \langle \sigma \rangle_{\mathit{store}}$$

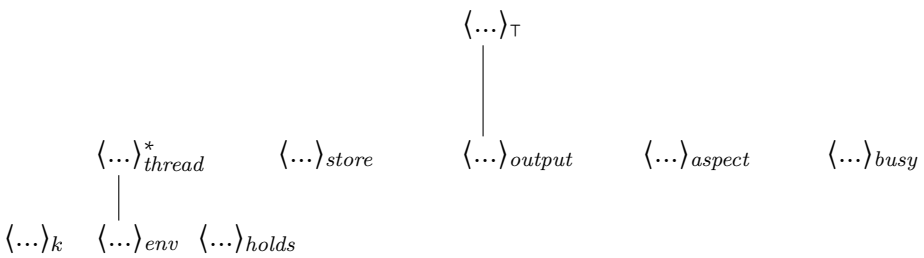
Since in the new configuration the store is located on a level different from that of the computation and the environment, this rule will never match with the new configuration structure. To fix this problem, one would apparently have to revisit the existing definitions and modify them to account for the new configuration structure, for example:

$$\langle \cdots \langle \frac{x}{\sigma[\rho[x]]} \cdots \rangle_k \ \langle \rho \rangle_{\mathit{env}} \cdots \rangle_{\mathit{thread}} \ \langle \sigma \rangle_{\mathit{store}}$$

This is of course very inconvenient and violates one of our major requirements for an ideal language definitional framework: modularity. How can we derive the second rule above from the first? K provides a mechanism called *context transforming*, which is responsible for automatically completing "partially" defined terms. This is explained in detail in [38]. In short, cell wrappers can be defined as "structural" and possibly "repetitive" in K; this additional information can be then used to automatically and unambiguously[1] transform terms in general, and K rules and equations in particular, by completing them with potentially missing structural information. For example, all the cells in this language definition are defined as "structural" and $\langle ... \rangle_{\mathit{thread}}$ is also defined as "repetitive". When using the algebraic notation for K as in [38], the language designer can add this information rigorously, using operation attributes to the cell wrapper constructs. We have not yet

[1] In case of ambiguity, the most "local" grouping is always chosen, following a depth-first traversal of the term.

devised any rigorous non-algebraic way to introduce this configuration structural information; for the remaining of this section we just draw a picture like the one below showing the configuration structure together with the repetitiveness information (a star "$*$" as a superscript of a cell means that the cell is allowed to repeat) and let the reader mentally replay the context transforming process:



The context transforming process is not only necessary in order to achieve the much desired modularity, but it also allows the designer to write semantic rules and equations more succinctly and conceptually: one only needs to focus on *what* is needed from the configuration in order to give the semantics of a construct, rather than *where* that information is located in the (continuously evolving) configuration structure. Thanks to context transforming, we need to make *no change* to the existing rules or equations when adding threads to our language. All we need to do is to add the definitions of the new constructs.

Here is the semantics of thread creation and termination:

$$\frac{\langle \underline{\mathsf{spawn}\ s}\ \cdots\rangle_k\ \langle\rho\rangle_{env}}{\cdot}\ \frac{\cdot}{\langle\langle s\rangle_k\ \langle\rho\rangle_{env}\ \langle\cdot\rangle_{holds}\rangle_{thread}}$$

$$\frac{\langle\cdots\ \langle\cdot\rangle_k\ \langle lc\rangle_{holds}\ \cdots\rangle_{thread}}{\cdot}\ \langle\ \frac{ls}{ls - lc}\ \rangle_{busy}$$

Here is the semantics of lock acquire:

$$\langle \frac{\mathsf{acquire}\ v}{\cdot}\ \cdots\rangle_k\ \langle\cdots\ (v, \frac{n}{s(n)})\ \cdots\rangle_{holds}$$

$$\langle \frac{\mathsf{acquire}\ v}{\cdot}\ \cdots\rangle_k\ \langle\cdots\ \frac{\cdot}{(v,0)}\ \cdots\rangle_{holds}\ \langle\ \frac{ls}{ls\ v}\ \rangle_{busy}\ \texttt{when}\ v \notin ls$$

Finally, here is the semantics of lock release:

$$\langle \frac{\mathsf{release}\,v}{\cdot}\ \cdots\rangle_k\ \langle\cdots\ (v, \frac{s(n)}{n})\ \cdots\rangle_{holds}$$

$$\langle \frac{\mathsf{release}\,v}{\cdot}\ \cdots\rangle_k\ \langle\cdots\ \frac{(v,0)}{\cdot}\ \cdots\rangle_{holds}\ \langle\cdots\ \frac{v}{\cdot}\ \cdots\rangle_{busy}$$

An interesting question is what is the resulting semantics of halt in this multi-threaded extension of our language, that is, whether it halts only the current thread or the entire program. Since its previous semantics was to just dissolve the computation that generated it, the answer is that it only halts the current thread. This is quite an acceptable semantics. Supposing that one wants a semantics that stops the entire multi-threaded program, then one has to remove the current rule for halt and add the following instead:

$$\langle\cdots\ \langle\mathsf{halt}\ \cdots\rangle_k\ \langle vl\rangle_{output}\ \cdots\rangle_\top \to vl$$

Variant 11 – rendez-vous synchronization. Let us next define a rendez-vous synchronization construct, say rv v, taking also a "lock", or better say a "barrier" argument. The thread executing this command is blocked until another thread executes a similar rendez-vous command with the same value v. When that happens, the two threads discard their rv v statements and continue their executions concurrently. Here is the semantics of rendez-vous synchronization in K:

$Stmt ::= \ldots \mid \mathsf{rv}\ \mathcal{E}\ [strict]$

$\langle \underset{\cdot}{\mathsf{rv}\ v} \cdots \rangle_k\ \langle \underset{\cdot}{\mathsf{rv}\ v} \cdots \rangle_k$

Note that, according to the structural and repetitiveness information about the configuration, the only way for the context transforming process to complete the rule above to match the configuration is as follows:

$\langle \cdots \langle \underset{\cdot}{\mathsf{rv}\ v} \cdots \rangle_k \cdots \rangle_{thread}\ \langle \cdots \langle \underset{\cdot}{\mathsf{rv}\ v} \cdots \rangle_k \cdots \rangle_{thread}$

Variant 12 – distributed agents with message communication. Let us next add distributed agents to the language, where each agent encapsulates a multi-threaded program with a locally shared store and where agents communicate with each other by both asynchronous and synchronous messages. Agents can create other agents and messages can contain any value, including other agent names to be used for sending or receiving messages. Here is the additional syntax:

$Agent ::=$ agent identifiers or names
$Val ::= \ldots \mid Agent$
$\mathcal{E} ::= \ldots \mid \mathsf{new\text{-}agent}\ Stmt \mid \mathsf{receive\text{-}from}\ \mathcal{E}\ [strict] \mid \mathsf{receive} \mid \mathsf{me} \mid \mathsf{parent}$
$Stmt ::= \ldots \mid \mathsf{send\text{-}asynch}\ \mathcal{E}\ \mathcal{E}\ [strict] \mid \mathsf{send\text{-}synch}\ \mathcal{E}\ \mathcal{E}\ [strict]$

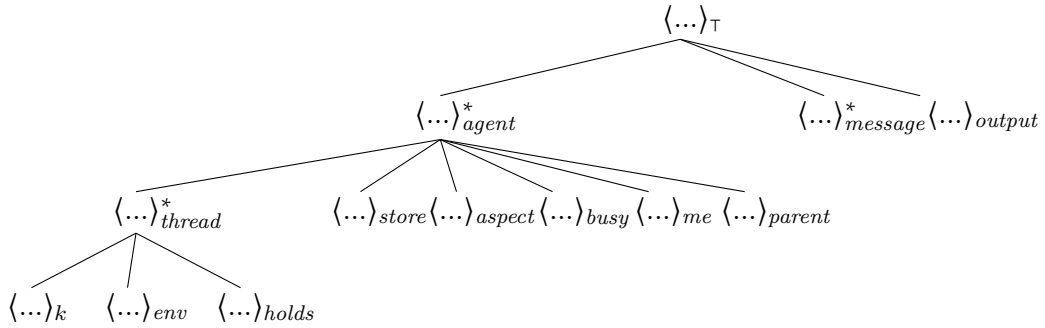
Agent is therefore a set set of agent names, which are regarded as any other values in the language, so that they can be passed and returned by functions, assigned to variables, send and received by messages, etc. new-agent s creates a new agent which will execute statement s concurrently with the rest of the agents (and the multiple threads inside those). Unlike in π-calculus (see Section **??**), we here disallow nested agents: all agents are located at the top. Communication between agents is exclusively via messages, which can be send asynchronously or synchronously. Messages contain a sender name, a receiver name, and a value. receive-from e first evaluates e to an agent name, say a, and then the receiving thread blocks until a message from a is received; when that happens, the thread is unblocked and the received value is passed to it. receive is similar, except that the receiving thread will accept any message sent to its enclosing agent. The special expression constructs me and parent evaluate to the current agent's name and to its parent's, respectively; this way, each agent is given access to two basic communication capabilities (access to more agents can be granted dynamically by other agents, by sending agent names via messages). send-asynch e_1 e_2 evaluates e_1 to an agent, say a, and e_2 to a value, say v, sends a message to a containing a, and then dissolves letting the execution to continue normally. send-synch is similar but it blocks the current thread until (one of the threads in) the destination agent receives the message.

An agent terminates when it has no thread left, and a program terminates when it has no agent left. Here is the new configuration syntax, together with its structural/repetition information, as well as the initialization and termination equations (when no agents left, remaining messages are discarded):

$Config ::= \ldots \mid \langle \mathsf{Set}[Config] \rangle_{agent} \mid \langle Agent, Agent, Val \rangle_{message}$

$[\![s]\!] = \langle \langle \langle s \rangle_k \ \langle \cdot \rangle_{env} \ \langle \cdot \rangle_{holds} \rangle_{thread} \ \langle \cdot \rangle_{store} \ \ \langle \cdot \rangle_{aspect} \ \langle \cdot \rangle_{busy} \ \langle n \rangle_{me} \ \langle n \rangle_{parent} \rangle_{agent} \ \langle \cdot \rangle_{output} \rangle_{\top}$ where $n \in Agent$ fresh

$\langle \langle vl \rangle_{output} \ M \rangle_{\top} \to vl$ when M contains only messages (zero or more)



Note that the output is shared by all agents and that there can be multiple agents and messages floating in the top-level soup, same way multiple threads can float in each agent. The following K rules are straightforward; they do formally precisely what was described informally above. Recall that context transforming is assumed to complete the partial configuration structural information in these rules:

$$\langle \underline{\mathsf{new\text{-}agent}\ s\ \cdots}_{m} \rangle_k \ \langle n \rangle_{me} \quad \frac{\cdot}{\langle \langle \langle s \rangle_k \ \langle \cdot \rangle_{env} \ \langle \cdot \rangle_{holds} \rangle_{thread} \ \langle \cdot \rangle_{store} \ \ \langle \cdot \rangle_{aspect} \ \langle \cdot \rangle_{busy} \ \langle m \rangle_{me} \ \langle n \rangle_{parent} \rangle_{agent}}$$ where $m \in A$

$\langle A \rangle_{agent} \to \cdot$ when A contains no thread

$$\langle \underline{\mathsf{me}\ \cdots}_{n} \rangle_k \ \langle n \rangle_{me}$$

$$\langle \underline{\mathsf{parent}\ \cdots}_{n} \rangle_k \ \langle n \rangle_{parent}$$

$$\langle \underline{\mathsf{send\text{-}asynch}\ m\ v\ \cdots}_{\cdot} \rangle_k \ \langle n \rangle_{me} \ \frac{\cdot}{\langle n, m, v \rangle_{message}}$$

$$\langle \underline{\mathsf{receive\text{-}from}\ n\ \cdots}_{v} \rangle_k \ \langle m \rangle_{me} \ \underline{\langle n, m, v \rangle_{message}}_{\cdot}$$

$$\langle \underline{\mathsf{receive}\ \cdots}_{v} \rangle_k \ \langle m \rangle_{me} \ \underline{\langle \_, m, v \rangle_{message}}_{\cdot}$$

$$\langle \cdots \ \langle \underline{\mathsf{send\text{-}synch}\ m\ v\ \cdots}_{\cdot} \rangle_k \ \langle n \rangle_{me} \ \cdots \rangle_{agent} \ \langle \cdots \ \langle \underline{\mathsf{receive\text{-}from}\ n\ \cdots}_{v} \rangle_k \ \langle m \rangle_{me} \ \cdots \rangle_{agent}$$

$$\langle \underline{\mathsf{send\text{-}synch}\ m\ v\ \cdots}_{\cdot} \rangle_k \ \langle \cdots \ \langle \underline{\mathsf{receive}\ \cdots}_{v} \rangle_k \ \langle m \rangle_{me} \ \cdots \rangle_{agent}$$

None of the definitions of the existing features needs to change. It is interesting, again, to analyze the resulting semantics of halt. If halt was defined as in "Variant 3" (extension with output),

then it would just dissolve the remaining computation in the current thread in the current agent. If halt was defined like in the alternative rule proposed at the end of "Variant 10" (extension with concurrency), then it would halt the entire program. Therefore, the two semantic alternatives for halt have extreme behaviors. For this language, it may make sense to have a halt statement that halts only the issuing agent, dissolving all its threads but allowing the remaining agents to continue their executions. If that is what one wants, then one needs to remove the previous rule of halt and add the following rule instead:

$$\langle \cdots \langle \mathsf{halt}\ \cdots \rangle_k\ \cdots \rangle_{agent} \to \cdot$$

Variant 13 – self-generation of code. There is an increasingly broad interest in generative programming these days. We next add support for dynamic code generation to our language.

$$\mathcal{E} ::= \ldots \mid \mathsf{quote}\ \mathcal{E} \mid \mathsf{unquote}\ \mathcal{E} \mid \mathsf{eval}\ \mathcal{E}\ [strict]$$
$$Val ::= \ldots \mid code(K)$$
$$K ::= \ldots \mid quote(Nat, \mathsf{List}[K]) \mid code(\mathsf{List}[K]) \mid K \curvearrowright K\ [strict] \mid \boxed{Id}(\mathsf{List}[K])\ [strict(2)] \mid K \curvearrowright K\ [strict]$$

$$\langle \mathsf{quote}(k)\ \cdots \rangle_k = \langle quote(0,k)\ \cdots \rangle_k$$
$$quote(n, k_1 \curvearrowright k_2) = quote(n,k_1) \curvearrowright quote(n,k_2) \qquad code(k_1) \curvearrowright code(k_2) = code(k_1 \curvearrowright k_2)$$
$$quote(n, f(kl)) = \boxed{f}(quote(n,kl)) \text{ if } f \neq \mathsf{quote}, \mathsf{unquote} \qquad \boxed{f}(code(kl)) = code(f(kl))$$
$$quote(n, \mathsf{quote}(k)) = \boxed{\mathsf{quote}}(quote(s(n),k))$$
$$quote(0, \mathsf{unquote}(k)) = k$$
$$quote(s(n), \mathsf{unquote}(k)) = \boxed{\mathsf{unquote}}(quote(n,k))$$

$$quote(n,(k,kl)) = quote(n,k) \curvearrowright quote(n,kl) \text{ if } kl \neq \cdot \qquad code(k) \curvearrowright code(kl) = code(k,kl)$$
$$quote(n,k) = code(k) \text{ if } k \in Val \cup Var \qquad \mathsf{eval}\ code(k) = k$$